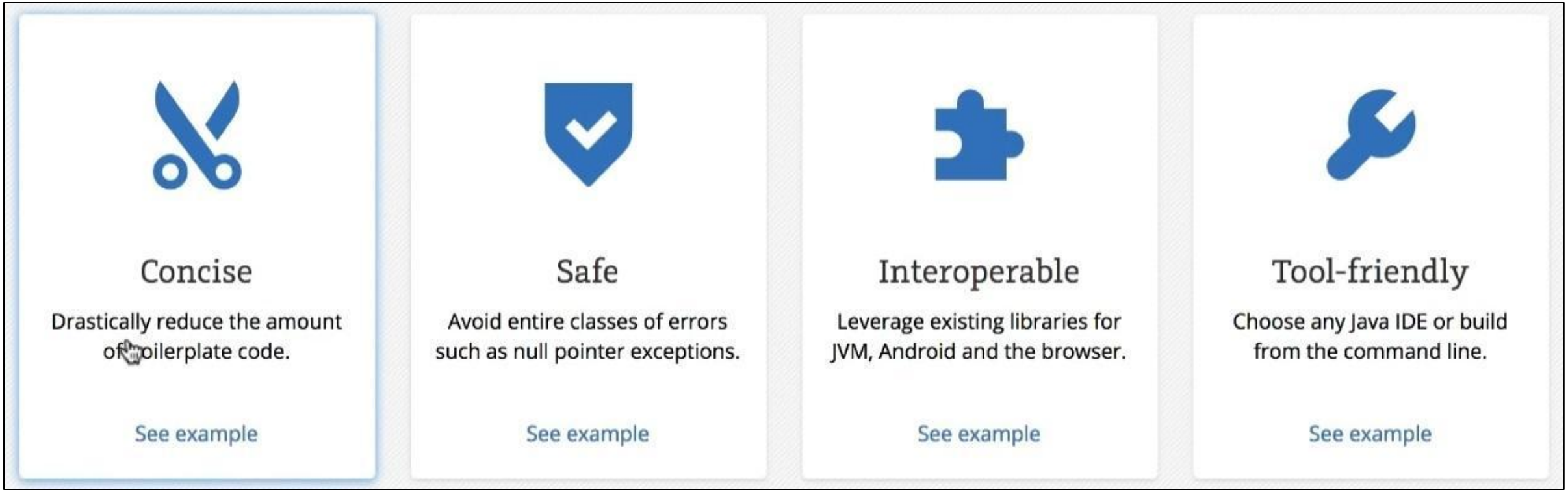
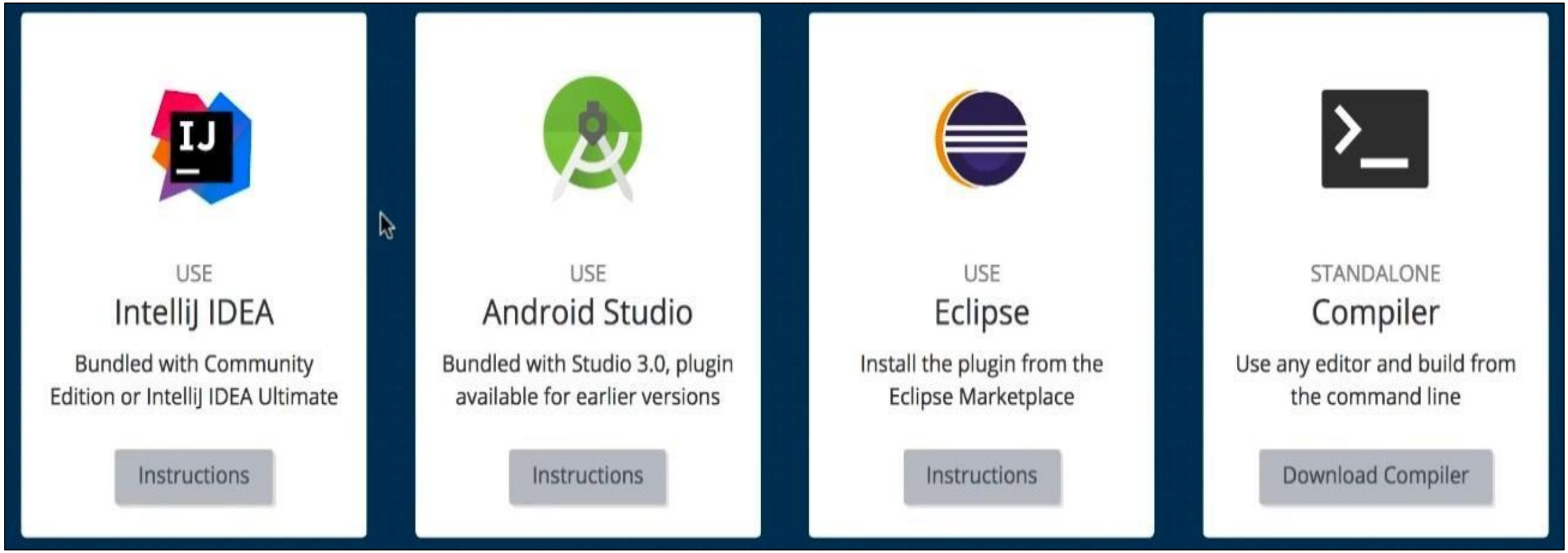
Introduction to Kotlin

Why Kotlin?


Concise
Drastically reduce the amount of boilerplate code.
See example
Safe
Avoid entire classes of errors such as null pointer exceptions.
See example
Interoperable
Leverage existing libraries for JVM, Android and the browser.
See example
Tool-friendly
Choose any Java IDE or build from the command line.
See example

Tool


USE
IntelliJ IDEA
Bundled with Community Edition or IntelliJ IDEA Ultimate
Instructions
USE
Android Studio
Bundled with Studio 3.0, plugin available for earlier versions
Instructions
USE
Eclipse
Install the plugin from the Eclipse Marketplace
Instructions
STANDALONE
Compiler
Use any editor and build from the command line
Download Compiler

Operator

Program

```
fun main(args:
Array<String>)
{
    var num1 : Int = 4
    var num2 : Int = 7
    var num3 = num1 +
num2
    var a1 = anand()
    println("addtion
of two number $num1
and $num2 is : $num3")
    a1.name = "anand"
    println("name
${a1.name}")
}
```

```
class anand {
    var name:
String? = null
}
```

Output

```
addtion of two number 4 and 7
is : 11
name anand
```

If else, else if ladder, as expression

Program

```
fun main(args :
Array<String>)
{
    var num1: Int
= 4
    var num2: Int
= 4

    var result :
Int = 0

    result =
if(num1 > num2)
        num1
    else if (num2
> num1)
        num2
    else
        0
    print(result)
}
```

Output

```
0
```

Class

Program

anand.kt

```
class anand {
    var name:
String? = null
}
```

Another class

```
fun main(args:
Array<String>)
{
    var a1 = anand()

    a1.name = "anand"

    print(a1.name)
}
```

Output

```
anand
```

Null Handling

Program

anand.kt

```
class anand {
    var name:
String? = null
}
```

```
fun main()
{
//    var str : String = null
//error
    var str1 : String? = null
    //print(str1)
    var a1 = anand()
//    print(a1.name.length) //
error
    println(a1.name?.length)
//    var a2:anand = anand()
//    a2 = null // error

    var a3:anand? = anand()
    a3 = null
    println(a3?.name)
}
```

Output

```
null
null
```

When

Program

```
fun main()
{
    var num : Int = 5
    when(num)
    {
        1 →
print("one")
        2 →
print("two")
        3 →
print("three")
        else →
print("invalid")
    }

}
```

Output

```
invalid
```

When as Expression

Program

```
fun main()
{
    var num : Int = 1

    var str = when(num)
    {
        1 → "one"
        2 → "two"
        3 → "three"
        else → "invalid"
    }
    print(str)
}
```

Output

```
one
```

Loop

Program

```
fun main(args:
Array<String>)
{
    var a: Int = 1
    while(a ⩽ 10)
    {
        println(a)
        a++
    }
}
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

Loop

Program

```
var a: Int =
1
do
{

println(a)
    a++
}while(a ⩽ -1)
```

Output

```
1
```

Loop

Program

```
var d = 1 .. 6
for (i in d)
{
    println(i)
}
```

Output

```
1
2
3
4
5
6
```

Loop

Program

```
var d = 1 .. 6
for (i in d
step 2)
{
    println(i)
}
```

Output

```
1
3
5
```

Loop

Program

```
var d = 1 .. 6
for (i in d.reversed())
{
    println(i)
}
```

Output

```
6
5
4
3
2
1
```

Loop

Program

```
var d = 6 downTo  1
for (i in d)
{
    println(i)
}
```

Output

```
6
5
4
3
2
1
```

Loop

Program

```
var d = 1 until  6
for (i in d)
{
    println(i)
}
println("count is
:"+d.count())
```

Output

```
1
2
3
4
5
count is :5
```

Loop

Program

```
var d = 'A' .. 'z'
for (i in d)
{
    println(i)
}
```

List

Program

```
fun main()
{
    var l = listOf(1,2,3,4)

    for (i in l)
    {
        println(i)
    }

}
```

Output

```
1
2
3
4
```

List

Program

```
fun main()
{
    var l = listOf(1,2,3,4)


    for ((index,i) in
l.withIndex())
    {
        println("on index [$index]
value is $i")
    }
}
```

Output

```
on index [0] value is
1
on index [1] value is
2
on index [2] value is
3
on index [3] value is
4
```

Map

Program

```
import java.util.*
fun main()
{
    var mapvar =
TreeMap<String,Int>()
    mapvar["anand"] = 100
    mapvar["janardan"] = 50

    for((name,mark) in mapvar)
    {
        println("$name : $mark")
    }
}
```

Output

```
anand : 100
janardan : 50
```

Function

Program

```
fun main()
{
    add(3,4)
}

fun add (a:Int,
b:Int)
{

println(a+b)
}
```

Output

```
7
```

Function

Program

```
fun main()
{
    var res = addr(5,6)
    println(res)
}
fun addr (a:Int,b:Int) : Int
{
    return a+b
}
```

Output

```
11
```

Function

Program

```
fun main()
{
    var res1 =
addrb(7,6)
    println(res1)
}
fun addrb
(a:Int,b:Int) : Int
= a+b
```

Output

```
13
```

Function

Program

```
fun main()
{
    var maxvalue =
max(7,9)
    println(maxvalue)
}
fun max (a:Int,b: Int)
: Int
{
    if(a>b)
        return a
    else
        return b
}
```

Output

```
9
```

Function

Program

```
fun main()
{
    var maxvalue1 =
max1(10,9)
    println(maxvalue1)
}
fun max1 (a:Int,b: Int) : Int
= if(a>b) a else b
```

Output

```
10
```

Function

Program

```
fun main()
{
    //named parameter
    var value = calculate(interest = 0.09,
amt = 100)
    println(value)
}
fun calculate(amt: Int, interest : Double =
0.03) : Int
{
    return (amt+amt*interest).toInt()
}
```

Output

```
109
```

Kotlin Function calling from java

Program

Kotlin file

```
fun main()
{
    //named parameter
    var value = calculate(interest =
0.09, amt = 100)
    println(value)
}
fun calculate(amt: Int, interest :
Double = 0.03) : Int
{
    return (amt+amt*interest).toInt()
}
@JvmOverloads
fun calculate1(amt: Int, interest :
Double = 0.03) : Int
{
    return (amt+amt*interest).toInt()
}
```

Java file

```
public class calldefaultfun {
    public static void main(String[]
args)
    {
        int res =
FundeaultparamKt.calculate(100,0.05);
        System.out.println(res);
        //int res1 =
FundeaultparamKt.calculate(100);//error
because two parameter should be provided

        int res2 =
FundeaultparamKt.calculate1(100);
        System.out.println(res2);
    }
}
```

Output

```
105
103
```

String to Int

Program

```
fun main()
{
    var str :
String = "4"
    var a : Int =
str.toInt()
    a++
    println(a)
}
```

Output

```
5
```

Exception Handling

Program

```
fun main()
{
    var str :  String = "4a"
    var a : Int = 0
    try {
        a = str.toInt()
    }
    catch (e :
NumberFormatException)
    {
        println("invalid input")
    }
    a++
    println(a)
}
```

Output

```
invalid input
1
```

Exception Handling

Program

```
fun main()
{
    var str :  String = "4a"
    var a : Int = try {
        str.toInt()
    }
    catch (e :
NumberFormatException)
    {
        7
    }
    a++
    println(a)
}
```

Output

```
8
```

Extension Function

Program

```
fun main()
{
    var prog1 = supportextfun()
    prog1.skills = "kotlin"
    println(prog1.skills)
    var prog2 = supportextfun()
    prog2.skills = "java"
    println(prog2.skills)
    var prog3 = prog1.plus(prog2)
    println(prog3.skills)
}
fun
supportextfun.plus(prog:supportext
fun) : supportextfun
{
    var prognew = supportextfun()
    prognew.skills = this.skills +
" " + prog.skills
    return prognew
}
```

```
class supportextfun {
    var skills : String?
= null

    fun show()
    {
        println(skills)
    }
}
```

Output

```
kotlin
java
kotlin java
```

Extension Function

Program

```
fun main()
{
    var prog1 = supportextfun()
    prog1.skills = "kotlin"
    println(prog1.skills)

    var prog2 = supportextfun()
    prog2.skills = "java"
    println(prog2.skills)

    var prog3 = prog1 plus prog2
    println(prog3.skills)
}

//infix keyword
infix fun
supportextfun.plus(prog:supportextfun) :
supportextfun
{
    var prognew = supportextfun()
    prognew.skills = this.skills + " " +
prog.skills
    return prognew
```

```
class supportextfun {
    var skills :
String? = null

    fun show()
    {
        println(skills)
    }
}
```

Output

```
kotlin
java
kotlin java
```

Extension Function

Program

```
fun main()
{
    var prog1 = supportextfun()
    prog1.skills = "kotlin"
    println(prog1.skills)

    var prog2 = supportextfun()
    prog2.skills = "java"
    println(prog2.skills)

    var prog3 = prog1 + prog2
    println(prog3.skills)
}

// operator overloading
operator fun
supportextfun.plus(prog:supportextfun) :
supportextfun
{
    var prognew = supportextfun()
    prognew.skills = this.skills + " " +
prog.skills
    return prognew
```

```
class supportextfun {
    var skills :
String? = null

    fun show()
    {

println(skills)
    }
}
```

Output

```
kotlin
java
kotlin java
```

Factorial

Program

```
fun main()
{
    var num = 5
    var fact = 1
    for (i in 1..num)
    {
        fact = fact * i
    }
    println("factorial of
$num is: "+fact)
}
```

Output

```
factorial of 5 is:
120
```

Recursion

Program

```
fun main()
{
    var num = 5 // upto 31 will give
correct answer
    println(fact(num))
}
fun fact (num : Int) : Int
{
    if(num == 0)
        return 1
    else
        return num * fact(num-1)
}
```

Output

```
120
```

Recursion

Program

```
import java.math.BigInteger

fun main()
{
    var num  = BigInteger("5") // up to 4777 will
give correct answer
    println(fact(num))
}
fun fact (num : BigInteger) : BigInteger
{
    if(num == BigInteger.ZERO)
        return BigInteger.ONE
    else {
//         println("hello : "+num)
        return num * fact(num - BigInteger.ONE)
    }
}
```

Output

```
120
```

Recursion

Program

```
import java.math.BigInteger


fun main()
{
    var num  = BigInteger(“5")
    println(fact(num, BigInteger.ONE))
}
tailrec fun fact (num : BigInteger, res :
BigInteger) : BigInteger
{
    if(num == BigInteger.ZERO)
        return res
    else {
        //println("hello : "+num)
        return fact(num -
BigInteger.ONE,num*res)
    }
}
```

Output

```
120
```

Constructor

Program

```
//primary constructor
class abc constructor(n: String = "janardan") //constructor keyword is optional
{
    var name : String = n
    fun show()
    {
        println("ABC $name")
    }
}
fun main()
{
    var a = abc("anand")
    a.show()
}
```

Output

```
ABC anand
```

Constructor

Program

```
//init block
class abc constructor(n: String) //constructor
keyword is optional
{
    var name : String = ""
    init {
        name = n
        println("init block")
    }
    fun show()
    {
        println("ABC $name")
    }
}
fun main()
{
    var a = abc("anand")
    a.show()
}
```

Output

```
init block
ABC anand
```

Constructor

Program

```
// secondary constructor
class abc constructor(n: String) //constructor
keyword is optional
{
    var name : String = ""
    var a : Int = 0
    constructor(a: Int,name: String) : this(name)
    {
        this. a = a
        this.name = name
    }
    fun show()
    {
        println("ABC $name : $a")
    }
}
fun main()
{
    var a = abc(10,"anand")
    a.show()
}
```

Output

```
ABC anand : 10
```

Inheritance

Program

```
open class A
{
    fun show()
    {
        println("class A
show method")
    }
}
class B : A()
{

}
fun main()
{
    var a = B()
    a.show()
}
```

Output

```
class A show method
```

Inheritance

Program

```
open class A
{
    fun show()
    {
        println("class A show method")
    }
}
open class C
{

}
class B : A(),C() //multiple
inheretance not supported
{

}
fun main()
{
    var a = B()
    a.show()
}
```

Output

```
Only one class may appear in a
supertype list
```

Inheritance

Program

```
open class A
{
    open fun show()
    {
        println("class A
show method")
    }
}
open class B : A()
{
    override fun show()
    {
        println("class B
show method")
    }
}

fun main()
{
    var a = B()
    a.show()
}
```

Output

```
class B show method
```

Inheritance

Program

```
open class A
{
    open fun show()
    {
        println("class A show method")
    }
}
open class B : A()
{
    override fun show()
    {
        println("class B show method")
    }
}

fun main()
{
    var a:A = B() // created object of B and
reference of A class
    a.show()
}
```

Output

```
class B show method
```

Constructor in Inheritance

Program

```
open class Aa
{
    init {
        println("Class Aa
init block")
    }
    open fun show()
    {
        println("class Aa
show method")
    }
}
open class Bb : Aa()
{
    init {
        println("Class Ba
init block")
    }
    override fun show()
    {
        println("class Bb
show method")
    }
```

```
fun main()
{
    var a
= Bb()

a.show()
}
```

Output

```
Class Aa init block
Class Ba init block
class Bb show
method
```

Constructor in Inheritance

Program

```
open class Aa (data : String)
{
    init {
        println("Class Aa init block "+data)
    }
    open fun show()
    {
        println("class Aa show method")
    }
}
open class Bb : Aa("hi")
{
    init {
        println("Class Ba init block")
    }
    override fun show()
    {
        println("class Bb show method")
    }
```

```
fun main()
{
    var a = Bb()

a.show()
}
```

Output

```
Class Aa init block
hi
Class Ba init block
class Bb show method
```

Constructor in Inheritance

Program

```
open class Aa (data : String)
{
    init {
        println("Class Aa init block "+data)
    }
    open fun show()
    {
        println("class Aa show method")
    }
}
open class Bb(d: String) : Aa(d)
{
    init {
        println("Class Ba init block")
    }
    override fun show()
    {
        println("class Bb show method")
    }
}
```

```
fun main()
{
    var a = Bb("hello")
    a.show()
}
```

Output

```
Class Aa init block hello
Class Ba init block
class Bb show method
```

Abstract Class

Program

```
abstract class ABC
{
    abstract fun display()
    fun show()
    {
        println("class Abc show method")
    }
}
class xyz : ABC()
{
    override fun display()
    {
        println("class xyz display method")
    }
}
```

```
fun main()
{
    var a = xyz()

a.display()

a.show()
}
```

Output

```
class xyz display method
class Abc show method
```

Interface

Program

```
interface ABCD
{
    fun show()
}
interface WXYZ
{
    fun display()
    fun abc()
    {
        println("abc in WXYZ
interface")
    }
}
class p : ABCD,WXYZ
{
    override fun show() {
        println("in P class
show method")
    }

    override fun display() {
        println("in P class
display method")
    }
```

```
fun main()
{
    var p1
= p()

p1.show()

p1.display
()

p1.abc()
}
```

Output

```
in P class show method
in P class display
method
abc in WXYZ interface
```

Interface

Program

```
interface ABCD
{
    fun show()
    fun abc()
    {
        println("abc in ABCD
interface")
    }
}
interface WXYZ
{
    fun display()
    fun abc()
    {
        println("abc in WXYZ
interface")
    }
}
```

```
class p : ABCD,WXYZ
{
    override fun show() {
        println("in P class
show method")
    }

    override fun display() {
        println("in P class
display method")
    }
}
fun main()
{
    var p1 = p()
    p1.show()
    p1.display()
    p1.abc()
}
```

Output

```
Class 'p' must
override public open
fun abc(): Unit
defined in
com.example.kotlin_c
ore.ABCD because it
inherits multiple
interface methods of
it
```

Interface

Program

```
interface ABCD
{
    fun show()
    fun abc()
    {
        println("abc in ABCD interface")
    }
}
interface WXYZ
{
    fun display()
    fun abc()
    {
        println("abc in WXYZ interface")
    }
}
```

```
class p : ABCD,WXYZ
{
    override fun show() {
        println("in P class show method")
    }

    override fun display() {
        println("in P class display method")
    }
    override fun abc()
    {
        println("abc in p class")
    }
}
fun main()
{
    var p1 = p()
    p1.show()
    p1.display()
    p1.abc()
}
```

Output

```
in P class show method
in P class display method
abc in p class
```

Interface

Program

```
interface ABCD
{
    fun show()
    fun abc()
    {
        println("abc in ABCD interface")
    }
}
interface WXYZ
{
    fun display()
    fun abc()
    {
        println("abc in WXYZ interface")
    }
}
class p : ABCD,WXYZ
{
    override fun show() {
        println("in P class show method")
    }

    override fun display() {
        println("in P class display method")
    }
    override fun abc()
    {
        super<ABCD>.abc()
        println("abc in p class")
    }
}
fun main()
{
    var p1 = p()
    p1.show()
    p1.display()
    p1.abc()
}
```

Output

```
in P class show method
in P class display method
abc in ABCD interface
abc in p class
```

Data class

Program

```
// 1. Every class need toString() method
// 2. override equals and hashCode
// 3. copy
class laptop(brand: String, price:Int)
{
    fun show()
    {
        println("laptop")
    }
}
fun main()
{
    var a = laptop("hp",2000)
    println(a) // we are not getting value of brand
and price
    var b = laptop("hp",2000)
    println(b) // to print brand and price we need
toString() (point 1)
    //var c = a.copy() // will give error (point 3)
    println(a.equals(b)) // to compare object need two
method (point 2)
    println(a==b) // to compare object need two method
(point 2)
    //println(c) // will give error (point 3)
```

Output

```
com.example.kotlin_core.laptop@71b
e98f5
com.example.kotlin_core.laptop@6fa
dae5d
false
false
```

Data class

Program

```
data class laptop(var brand: String,var price:Int) // var or
val must be there
{
    fun show()
    {
        println("laptop")
    }
}
fun main()
{
    var a = laptop("hp",2000)
    println(a)
    var b = laptop("hp",2000)
    println(b)
    var c = a.copy()
    println(a.equals(b))
    println(a==b)
    println(c)
}
```

Output

```
laptop(brand=hp,
price=2000)
laptop(brand=hp,
price=2000)
true
true
laptop(brand=hp,
price=2000)
```

Program

Data class

```
data class laptop(var brand: String,var price:Int) // var
or val must be there
{
    fun show()
    {
        println("laptop")
    }
}
fun main()
{
     var a = laptop("hp",2000)
     println(a)
     var b = laptop("hp",2000)
     println(b)
     var c = a.copy(price = 2500)
     println(a.equals(b))
     println(a==b)
     println(c)
}
```

Output

```
laptop(brand=hp,
price=2000)
laptop(brand=hp,
price=2000)
true
true
laptop(brand=hp,
price=2500)
```

Object keyword

Program

```
object obj // class where only one object can be created
{
    var a = 50
    fun show()
    {
        println("value of a is:"+a)
    }
}
fun main()
{
    obj.a = 100 // no need to create object
    obj.show()
}
```

Output

```
value of a is:100
```

Object keyword

Program

```
data class car(var name: String, var
price : Int)
object garage
{
    var cars = arrayListOf<car>()
    fun showcar()
    {
        for (i in cars)
        {
            println(i)
        }
    }
}
fun main()
{

garage.cars.add(car("venue",500))

garage.cars.add(car("nexon",550))
    garage.showcar()
}
```

Output

```
car(name=venue, price=500)
car(name=nexon, price=550)
```

Anonymous inner class

Program

```
interface asd
{
    fun show()
}
fun main()
{
    var a = object :
asd{
        override fun
show() {

println("in show")
        }
    }
    a.show()
}
```

Output

```
in show
```

Companion Object

Program

```
class jk
{
    companion object
    {
        fun show()
        {

println("in show")
        }
    }
}
fun main()
{
    jk.show()
}
```

Output

```
in show
```

Companion Object (work like static)

Program

Kotlin file

```
class jk
{
    companion
object
    {
        fun show()
        {

println("in show")
        }
    }
}
fun main()
{
    jk.show()
}
```

Java file

```
public class companionobjectex
{
    public static void
main(String[] args)
    {
        jk.show();
    }
}
```

Output

```
error: cannot find
symbol
        jk.show();
          ^
  symbol:   method
show()
  location: class jk
```

Companion Object (work like static)

Program

Kotlin file

```
class jk
{
    companion
object
    {
        @JvmStatic
        fun show()
        {

println("in show")
        }
    }
}
fun main()
{
    jk.show()
}
```

Java file

```
public class companionobjectex
{
    public static void
main(String[] args)
    {
        jk.show();
    }
}
```

Output

```
in
show
```

Companion Object (factory design pattern)

Program

```
class compan
{
    companion object
    {
        fun create() : compan = compan()
    }
    fun show()
    {
        println("in show")
    }
}
fun main()
{
    //var obj = compan() // if sometime
hard to create obj
    var obj = compan.create()
    obj.show()
}
```

Output

```
in show
```

Array

Program

```
fun main()
{
    var nums =
intArrayOf(10,20,30,40)
    //println(nums) // will get
hashcode not elements
    //nums[1] = 25
    nums.set(1,25)
    for(i in nums)
        println(i)
    //println(nums.get(2))
    //println(nums[2])
    //println(nums.last())
    var num2 = IntArray(4)
    num2[0] = 70
    num2[1] = 71
    num2[2] = 72
    num2[3] = 73
    for(i in num2)
        println(i)
    var num3 =
arrayOf("A","B","C","D")
    for(i in num3)
        println(i)
    var num4 =
arrayOf(1,2,3,4)
    for(i in num4)
        println(i)
    var num5 =
arrayOfNulls<String>(5)
    num5[0] = "abc"
    println(num5[0])
}
```

Output

```
10
25
30
40
70
71
72
73
A
B
C
D
1
2
3
4
abc
```

List (immutable)

Program

```
fun main()
{
    var v =
listOf<Int>(7,8,9,0)

    for(i in v)
        println(i)
    //v.add() //give error as
list in immutable
    println(v.get(0))
    println(v.last())
}
```

Output

```
7
8
9
0
7
0
```

List (mutable)

Program

```
fun main()
{
    var v = mutableListOf<Int>(7,8,9,0)
    v.add(15)
    v.add(1,25)
    for(i in v)
        println(i)
}
```

Output

```
7
25
8
9
0
15
```

List of object

Program

```
data class ana(var name: String,var
mark:Int)
fun main()
{
    var a = listOf<ana>(ana("anand",70),
ana("abc",80))
    for(i in a) {
        println(i)
        println(i.name)
    }
}
```

Output

```
ana(name=anand, mark=70)
anand
ana(name=abc, mark=80)
abc
```

Type Checking

Program

```
class type {
}
fun main()
{
    var t = type()
    if(t is type) // type checking whether t
is type of "type" class
    {
        println("given object is of that
class")
    }
/*    if(p is type) //will give error as p is
not object of type class
    {
        println("given object is of that
class")
    }*/
}
```

Output

```
given object is of
that class
```

Package

Program

```
package com.example.kotlin_core.mypackage

class mypackageclass {
    fun show()
    {
        println("show method inside mypackage")
    }
}
```

```
import com.example.kotlin_core.mypackage.mypackageclass

fun main()
{
    var obj = mypackageclass()
    obj.show()
}
```

Output

```
show method inside mypackage
```

Safe call & elvis operator

Program

```
fun main()
{
    var gender: String? = null
    if(gender ≠ null)
        println(gender.length)
    println(gender?.length) // safe call operator: it
checked for gender is null or not, if not null then call
length on it otherwise will not call
    //if i want to execute multiple statement after
making sure that its not null
    gender?.let {
        println("hi")
        println("hi $gender")
        println("hi $it") // it will access gender value
    }
    var value = gender ?: "NA" // elvis operator: will
assign "NA" to value variable if its null otherwise will
assign value of gender to value variable
    println(value)
    var v = gender!!.length // not null asserted call
operator: will through error if its null otherwise store
length
```

Output

```
null
NA
Exception in thread
"main"
java.lang.NullPointerE
xception
    at
com.example.kotlin_cor
e.CallsafeandelvisopexKt.main(callsafeandelv
isopex.kt:17)
    at
com.example.kotlin_cor
e.CallsafeandelvisopexKt.main(callsafeandelv
isopex.kt)
```

Generics

Program

```
class gen<a>(var data:
a)
{
    fun getvalue() : a{
        return data
    }
}
fun main()
{
    var g1 = gen<Int>(3)

println(g1.getvalue())
    var g2 =
gen<String>("hello")

println(g2.getvalue())
}
```

Output

```
3
hello
```

Lambdas

```
fun sum(a: Int, b: Int): Int
{
    return a+b
}
```

```
val sum = {a:Int, b: Int -> a+b}
```

Lambdas

Program

```
fun main()
{
    var lambdas1 = {x: Int, y:
Int → x+y}
    println(lambdas1(2,3))

    var multilinelambdas = {

println("multilinelabmdas
called")
        println("line 2")
        var a= 2+3
        "hello"
    }
    multilinelambdas()

println(multilinelambdas())
}
```

Output

```
5
multilinelabmdas
called
line 2
multilinelabmdas
called
line 2
hello
```

Lambdas

Program

```
fun main()
{
    var singleparamlambdas = {x: Int → x*x}
    println(singleparamlambdas(2))
    var singleparamlambdas1:(Int)→Int = {x → x*x}
    println(singleparamlambdas1(3)) // its similar
to above one but here we have define type first so
no need to define x type

    var singleparam1 = {msg: String →
println("hello $msg")}
    singleparam1("anand")
    var singleparam2:(String) → Unit = {msg →
println("hello $msg")}
    singleparam2("anand") // its similar to above
one but here we have define type first so no need to
define msg type
}
```

Output

```
4
9
hello anand
hello anand
```

Lambdas

Program

```
fun main()
{
    var singleparam1:(Int)→Int = {x
→ x+x}
    println(singleparam1(4))
    var
simplyfiedsingleparam2:(Int)→Int =
{it + it}
    println(simplyfiedsingleparam2(3))
// its similar to above one but here
we have used default variable "it" so
no need to even write name
}
```

Output

```
8
6
```

Higher order function

Program

```
import java.util.function.Consumer
fun main()
{
    var v = listOf<Int>(1,2,3,4)
    var con : Consumer<Int> = object :
Consumer<Int>
    {
        override fun accept(t: Int) {
            println(t)
        }
    }
    v.forEach(con)
}
```

Output

```
1
2
3
4
```

Higher order function

Program

```
fun main()
{
    var v =
listOf<Int>(1,2,3,4)
    v.forEach({t →
println(t)})
    v.forEach({n →
println(n)})
}
```

Output

```
1
2
3
4
1
2
3
4
```

Higher order function

Program

```
fun main()
{
    var v =
listOf<Int>(1,2,3,4)

v.forEach({println(it
)})
}
```

Output

```
1
2
3
4
```

Higher order function

Program

```
fun main()
{
    var v = listOf<Int>(1,2,3,4)

v.forEach(::println)
}
```

Output

```
1
2
3
4
```

Filter & Map

Program

```
fun main()
{
    var v = listOf<Int>(1,2,3,4)
    var v1 = v.filter { it%2==0 }

v1.forEach({println(it)})
}
```

Output

```
2
4
```

Filter & Map

Program

```
fun main()
{
    var v = listOf<Int>(1,2,3,4)
    var v1 = v.filter { it%2==0 }
    var v2 = v1.map { it*2 }

v2.forEach({println(it)})
}
```

Output

```
4
8
```

Filter & Map

Program

```
fun main()
{
    var v =
listOf<Int>(1,2,3,4)
    var res = v.filter {
it%2==0 }.map { it*2 }
    res.forEach({ println(it)})
}
```

Output

```
2
4
```

Nested class

Program

```
class outer
{
    class nested
    {
        fun show()
        {
            println("nested class show method
inside outer class")
        }
    }
}
fun main()
{
    var nestobj = outer.nested()
    nestobj.show()
}
```

Output

```
nested class show
method inside outer
class
```

Nested class

Program

```
class outer
{
    var i = 0
    class nested
    {
        fun show()
        {
            println("nested class show method
inside outer class $i")
        }
    }
}
fun main()
{
    var nestobj = outer.nested()
    nestobj.show()
}
```

Output

```
Unresolved reference:
i
```

Inner class

Program

```
class outer
{
    var i = 0
    inner class innerclass
    {
        fun show()
        {
            println("innerclass show method
inside outer class $i")
        }
    }
}
fun main()
{
    val inn = outer().innerclass()
    inn.show()
}
```

Output

```
innerclass show method
inside outer class 0
```

Enum

Program

```
enum class color
{
    red,
    green,
    blue
}
fun main()
{
    color.values().forEach
{ println(it) }
}
```

Output

```
red
green
blue
```

Enum

Program

```
enum class color
{
    red,
    green,
    blue
}
fun main()
{
    var c = color.red
    when(c)
    {
        color.blue →
println("blue color")
        color.green →
println("green color")
        color.red →
println("red color")
    }
}
```

Output

```
red color
```

Enum

Program

```
enum class color(var colorname: String,var
colorvalue: Int)
{
    red("r",10),
    green("g",11),
    blue("b",12)
}
fun main()
{
    println(color.red.colorname)
    println(color.red.colorvalue)
}
```

Output

```
r
10
```

Program

Enum

```
interface docolor
{
    fun show()
}
enum class color(var colorname: String,var
colorvalue: Int) : docolor // only interface extends
not class
{
    red("r",10){
        override fun show() {
                        println("colored with red")
                    }
                },
    green("g",11){
        override fun show() {
            println("colored with green")
        }
    },
    blue("b",12){
        override fun show() {
            println("colored with blue")
        }
    }
}
```

```
fun main()
{

color.green.show
()
}
```

Output

```
colored with green
```

Enum

Program

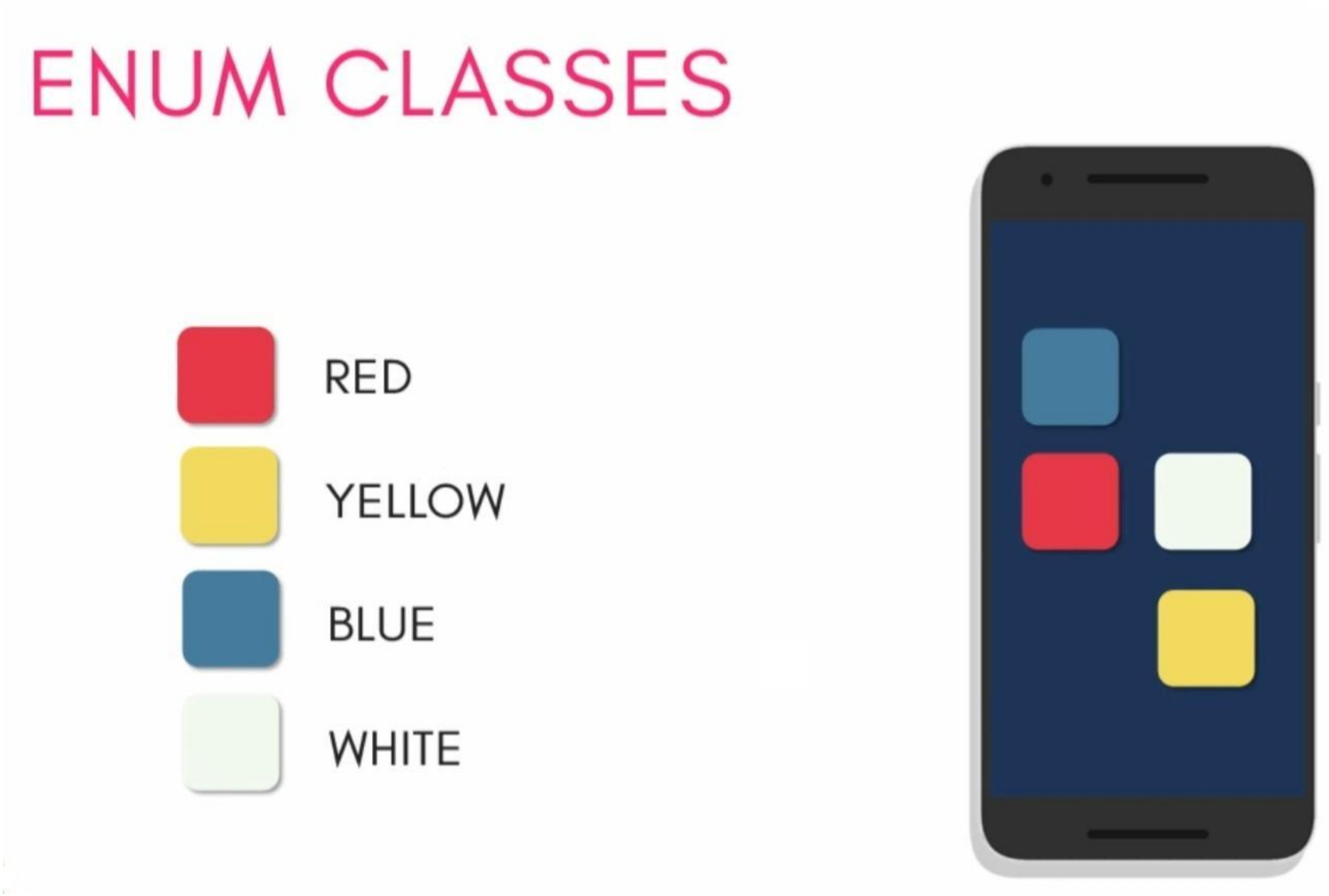
```
enum class day(val
number: Int)
{
    sunday(1),
    monday(2),
    tuesday(3),
    wednesday(4),
    thursday(5),
    friday(6),
    saturday(7);
    fun printday()
    {
        println("day is
$this")
    }
}
```

```
fun main()
{
    var da =
day.tuesday
    println(da)

println(da.number
)
    for( i in
day.values())
    {

println(i)
    }
    da.printday()
}
```

Output

```
tuesday
3
sunday
monday
tuesday
wednesday
thursday
friday
saturday
day is tuesday
```

Enum vs Sealed class



Enum vs Sealed class

SEALED CLASSES

Composition

Problem in Inheritance

- **We can only extend one class.** Extracting functionalities using inheritance often leads to either excessively complex hierarchies of types or to huge BaseXXX classes that accumulate many functionalities.
- **When we extend, we take everything from a class**, which leads to classes that have functionalities and methods they don't need (a violation of the Interface Segregation Principle).
- **Using superclass functionality is much less explicit.** In general, it is a bad sign when a developer reads a method and needs to jump into superclasses many times to understand how this method works.

Composition

Inheritance is a powerful feature, but it is designed to create a hierarchy of objects with an "is a" relationship. When such a relationship is not clear, inheritance might be problematic and dangerous. When all we need is a simple code extraction or reuse, inheritance should be used with caution; instead, we should prefer a lighter alternative: class composition.

Composition is the technique of creating a new class by combining existing classes. This is achieved by creating an instance of the existing class within the new class and delegating the required functionality to the instance. The composition can be used to achieve code reuse without creating complex inheritance hierarchies.

Delegation

Program

```
interface base
{
    fun show()
}
class baseimp : base
{
    override fun show() {
        println("in baseimp
show method")
    }
}
class derived : base by
baseimp()
fun main()
{
    var d = derived()
    d.show()
}
```

Output

```
in baseimp show method
```

Overriding a member of an interface implemented by delegation

Program

```
interface base
{
    fun show()
    fun show1()
}
class baseimp : base
{
    override fun show() {
        println("in baseimp show
method")
    }

    override fun show1() {
        println("in baseimp
show1 method")
    }
}
```

```
class derived : base by
baseimp()
{
    override fun show1() {
        println("in derived
show1 method")
    }
}
fun main()
{
    var d = derived()
    d.show()
    d.show1()
}
```

Output

```
in baseimp show
method
in derived show1
method
```

Overriding a member of an interface implemented by delegation

Program

```
interface base
{
    var msg: String
    fun show()
}
class baseimp : base
{
    override var msg = "hello"
    override fun show() {
        println("in baseimp show
method $msg")
    }
}
```

```
class derived : base by baseimp()
{
    // This property is not accessed
from baseimp implementation of
`show`
    override var msg = "hi"
}
fun main()
{
    var d = derived()
    d.show()
    println(d.msg)
}
```

Output

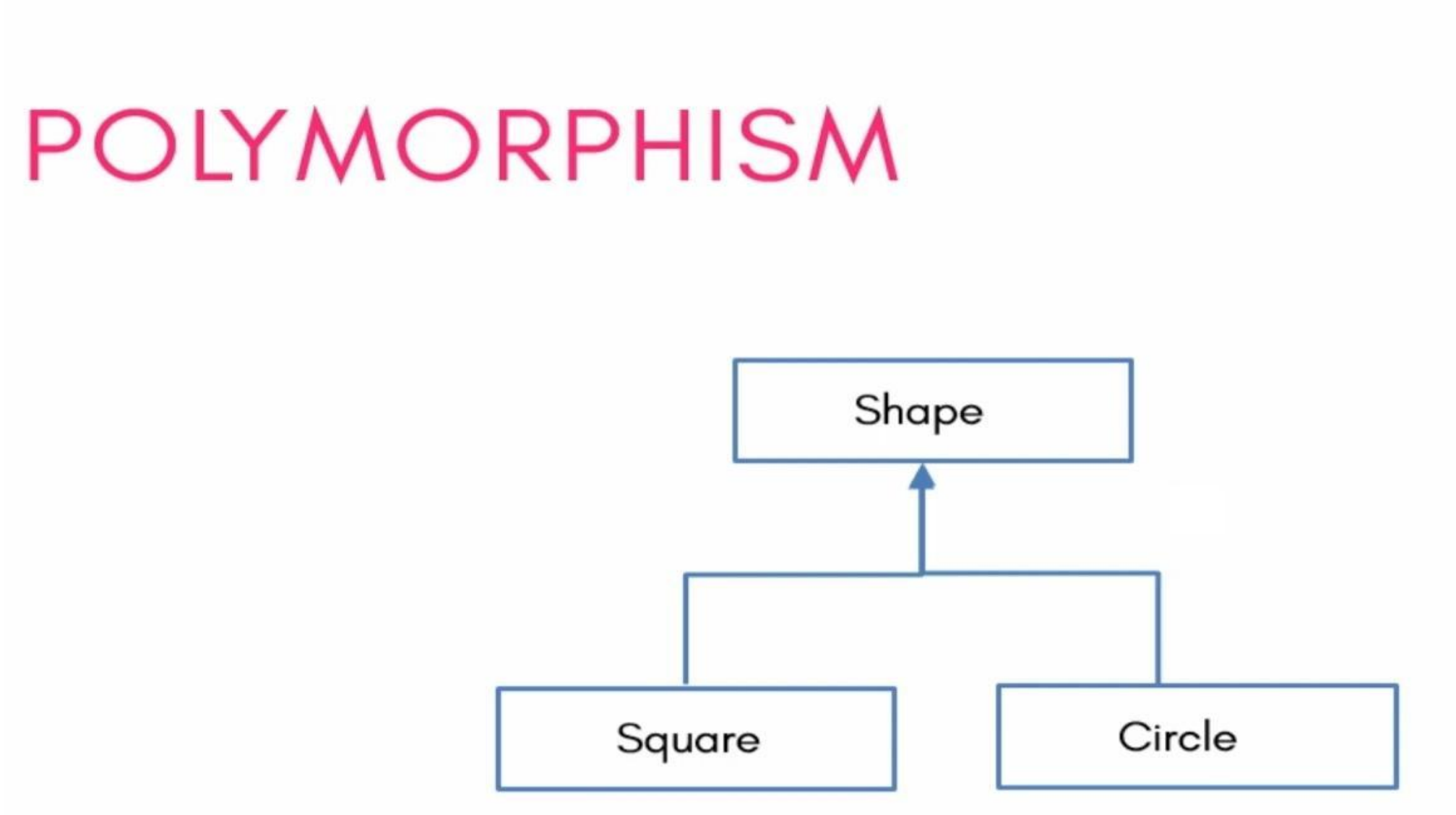
```
in baseimp show method
hello
hi
```

Polymorphism

POLYMORPHISM

- Parent can hold a reference to its child
- Parent can call methods of child classes (which are common)

Polymorphism



Polymorphism

Program

```
open class shape
{
    open fun area() : Double
    {
        return 0.0
    }
}
class circle(var radius:
Double) : shape()
{
    override fun area() :
Double
    {
        return Math.PI * radius
* radius
    }
}
```

```
class square(var side: Double)
: shape()
{
    override fun area() :
Double
    {
        return side * side
    }
}
fun main()
{
    var c:shape = circle(4.0)
    var s:shape = square(4.0)
    println(c.area())
    println(s.area())
}
```

Output

```
50.26548245743669
16.0
```

Polymorphism

Program

```
open class shape
{
    open fun area() : Double
    {
        return 0.0
    }
}
class circle(var radius:
Double) : shape()
{
    override fun area() :
Double
    {
        return Math.PI * radius
* radius
    }
}
class square(var side: Double) :
shape()
{
    override fun area() : Double
    {
        return side * side
    }
}
fun calculatearea(shapes: Array<shape>)
{
    for(s in shapes)
    {
        println(s.area())
    }
}
fun main()
{
    var c:shape = circle(4.0)
    var s:shape = square(4.0)
    var sa =
arrayOf(circle(3.0),circle(4.0),square(
5.0))
    calculatearea(sa)
}
```

Output

```
28.274333882308138
50.265482457436691
25.0
```

Up casting

What is Upcasting in Kotlin?

Upcasting is the idea of creating an object from a child class and storing that object into a variable that is of type super class!

Now the important thing to remember is that when an object of type child class is stored in a variable of type parent class, we can only access those members of the object that are shared between the child and parent class! This means if the object has created more functions in its body, then using a variable of parent class, we can't access those functions anymore.

Up casting

Program

```
open class wer
{
    fun show1()
    {
        println("in class wer show1 method")
    }
}
class rty : wer()
{
    fun show2()
    {
        println("in class rty show2 method")
    }
}
fun main()
{
    var obj:wer = rty() // upcasting
    obj.show1()
    //obj.show2() // will give error as show2 is
not part of class wer
}
```

Output

```
in class wer show1
method
```

Down casting

What is Downcasting in Kotlin?

So far, we've seen that when upcasting, only the shared members of both classes will be accessible using the variable of parent type.

But what if we've stored an object of a child type in a variable of type parent and then we wanted to cast (convert) that back into the child type?

Down Casting

Program

```
open class wer
{
    fun show1()
    {
        println("in class wer show1 method")
    }
}
class rty : wer()
{
    fun show2()
    {
        println("in class rty show2 method")
    }
}
```

```
fun main()
{
    var obj:wer = rty() // upcasting
    obj.show1()
    //var obj1: rty = obj // will give error type mismatch
    if (obj is rty)
    {
        obj.show2() // downcasting
    }
    //obj.show2()// will give error outside if body
}
```

Output

```
in class wer show1 method
in class rty show2 method
```

Down Casting

Program

```
open class wer
{
    fun show1()
    {
        println("in class wer
show1 method")
    }
}
class rty : wer()
{
    fun show2()
    {
        println("in class rty
show2 method")
    }
}
```

```
fun main()
{
    var obj:wer = rty() //
upcasting
    obj.show1()
    var obj1: rty = obj as rty
// downcasting
    obj1.show1()
    obj1.show2()
}
```

Output

```
in class wer show1
method
in class wer show1
method
in class rty show2
method
```